RL-TR-96-82
Final Technical Report
June 1996

EXPERIMENTATION WITH ADAPTIVE SECURITY POLICIES

Secure Computing Corporation

Edward A. Schneider, William Kalsow, Lynn TeWinkel, and Michael Carney

Rome Laboratory
Air Force Materiel Command
Rome, New York

APPROVED: Emilie J. Siarkiewicz

EMILIE J. SIARKIEWICZ
Project Engineer

FOR THE COMMANDER: [illegible]

JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE June 1996	3. REPORT TYPE AND DATES COVERED Final Dec 94 - Dec 95

4. TITLE AND SUBTITLE EXPERIMENTATION WITH ADAPTIVE SECURITY POLICIES	5. FUNDING NUMBERS C - F30602-95-C-0047 PE - 33140F PR - 7820 TA - 04 WU - 27
6. AUTHOR(S) Edward A. Schneider, William Kalsow, Lynn TeWinkel, and Michael Carney	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Secure Computing Corporation 2675 Long Lake Road Roseville MN 55113	8. PERFORMING ORGANIZATION REPORT NUMBER N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory/C3AB 525 Brooks Rd Rome NY 13441-4505	10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-96-82

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Emilie J. Siarkiewicz/C3AB/(315) 330-2135

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This work provides experimental evidence of the validity of recent theoretical work in the area of adaptive security policies. Solutions for several issues with these policies, including the coupling between policy and implementation, control over policy changes, stale cached data, reassigning security attributes, and recovery from change, were examined using the Distributed Trusted Operating System. Dynamic security lattices and task-based access control, previously studied by ORA, were also examined. The issue of trade-offs between security and fault tolerance, raised by SRI, was also studied, especially the problems of adapting the policy of a fault-tolerant service. An adaptation of an MLS policy enhanced with Type enforcement to a similar policy with more permissions was demonstrated, first using a single Security Server in which the policy table is replaced, and then handing off security decisions from one Security Server to another. Exercise of the relaxed permissions was audited, as specified by the Security Server.

14. SUBJECT TERMS Computer security, Adaptive security policies, Distributed Trusted Operating System, Cache control, Auditing, Dynamic security lattices, Fault tolerance, Policy-enforcement separation	15. NUMBER OF PAGES 56
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Contents

1 Introduction 1
- 1.1 Special Issues of Adaptive Policies . . . 2
  - 1.1.1 Coupling of Policy and Implementation . . . 2
  - 1.1.2 Inappropriate Policy Changes . . . 2
  - 1.1.3 Stale Cached Security Data . . . 3
  - 1.1.4 Reassigning Security Attributes . . . 3
  - 1.1.5 Recovering from Changes . . . 3
- 1.2 Document Overview . . . 3

2 DTOS Overview 5

3 DTOS Updates 9
- 3.1 Changing Policies . . . 9
- 3.2 Cache Control . . . 13
- 3.3 Security Identifier to Context Mapping . . . 15
- 3.4 Auditing . . . 16
- 3.5 Conclusion . . . 17

4 Implementation Using DTOS 18
- 4.1 Audit . . . 18
- 4.2 Handoff . . . 21
- 4.3 Page Table Reset . . . 23

5 Assess Feasibility 25
- 5.1 Dynamic Security Lattices . . . 25
  - 5.1.1 Background . . . 25
  - 5.1.2 Using a Dynamic Security Lattice . . . 29
  - 5.1.3 Dynamic Security Lattice Implementation on the DTOS Architecture . . . 31
  - 5.1.4 Non-transitive Extensions to Dynamic Security Lattices . . . 33
- 5.2 Delegated Policies: A Specialized Form of Adaptive Policy . . . 34
- 5.3 Task-based Access Control Policies . . . 36

6 Fault Tolerance and Security **38**

6.1 Security Policy Differences as Faults 39

6.2 Policy Transitions 41

7 Conclusion **43**

7.1 Open Issues 43

Section 1

Introduction

A security policy is "the set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information" [15]. An adaptive security policy, then, is one in which the ways that an organization manages, protects, and distributes sensitive information change as the system evolves. Command and Control systems require adaptive policies to support trade-offs between system operational capabilities and security in real-world situations. Situations in which the security policy may need to adapt to better serve the needs of its users include:

Reconfiguration of System Resources. For example, if a server node fails, the system might be reconfigured so that some other node becomes the server and possibly gains privileges that it needs to provide service.

Change in Operational Mode. For example, suppose that scheduling for a reconnaissance flight is classified as a secret activity. During a battle, a directive based on top secret information to search a particular area might need to be given. As another example, a flight control system might ignore all security checks in light of the imminent destruction of the aircraft.

Such adaptability is essential in the context of defensive information warfare. For example, there must be some mechanism to change the policy to contain system processes that become subverted.

Theoretical work has been done regarding the identification of interesting adaptive policies and formalization of these policies [8, 18]. In addition, there have been recent Rome Laboratory efforts to:

- investigate methods for implementing, maintaining, and analyzing security in adaptive systems [9], and
- specify systems that must adapt to changing environmental conditions to meet mission objective [7].

However, little practical experience exists with regard to the feasibility of implementing adaptive security policies. The objective of the work reported here is to gain that experience, making use of the Distributed Trusted Operating System (DTOS) prototype.

1.1 Special Issues of Adaptive Policies

Security policies are typically implemented by assigning *security attributes* to processes and objects and defining access control *rules* in terms of the security attributes. For example, UNIX^{TM} enforces security by associating user and group attributes with processes and associating permission bits with files. As another example, MultiLevel Secure (MLS) systems associate sensitivity levels with processes and objects and control access based on these levels.

A change in a system's security policy might be as simple as changing the rules that define the policy, such as collapsing levels together in an MLS system. In this case, the security attributes associated with each entity are not changed. More radical policy changes might alter the kinds of attributes associated with entities, such as switching from implementing a simple UNIX-like policy to implementing an MLS policy when a single-level system is added to a multi-level network during a crisis. Subsequent references to adaptive policies should be assumed to apply to either type of policy change.

The existing theoretical work focuses on the identification of interesting adaptive security policies and formalization of those policies. Our goal was to determine the feasibility of implementing such policies. We first evaluate the DTOS design with respect to several issues that we had determined were important for adaptive security and modified a DTOS prototype based on this evaluation. We then looked at the theoretical work with respect to this refined DTOS design. An additional issue relating fault tolerance to adaptive security was also raised by the theoretical work. The issues originally selected are as follows:

1.1.1 Coupling of Policy and Implementation

Most computer security products are developed to implement a single security policy. In MLS systems, the level-based security rules are typically hard-coded into the system. Since MLS is inherent to these systems, changing the policy to something other than MLS is not practical. Such implementations provide little if any support for adaptive policies.

1.1.2 Inappropriate Policy Changes

The security policy in effect at a particular time specifies what objects in the system are protected and what accesses the various processes in the system are allowed to make to those objects. Thus, the wrong security policy can allow a process to acquire accesses that should be forbidden, or prevent a process from acquiring accesses necessary for its mission. Typically, the system security policy would be analyzed before deployment to ensure correctness.With adaptive policies, errors can be introduced by policy changes being made by unauthorized users or at inappropriate times.

1.1.3 Stale Cached Security Data

For efficiency reasons, system components that enforce a policy may cache policy information. Page tables may contain protection bits for allocated pages of memory. When the policy changes, cached security information may become incorrect. Invalidating such cached data can be prohibitively expensive [10].

1.1.4 Reassigning Security Attributes

For those adaptive policies in which the security attributes associated with entities are changed, provisions must be made for determining the new security attributes to assign to each entity. The possible changes include both values for specific attributes and the set of attributes that make up a context. For example, a node failure in a distributed system may require the assignment of levels to each entity of the node that takes over the failed node's responsibilities.

1.1.5 Recovering from Changes

A commonly hypothesized use of adaptive security policies is to temporarily weaken security mechanisms to provide greater availability. For example, military units under attack might be given access to classified intelligence data that they need to counter the attack. While the security mechanisms are weakened, the classified data might leak into unclassified files. When the stronger security mechanisms are enabled at the completion of the attack, they cannot adequately protect the data. Even though the stronger mechanism will prohibit users with a clearance of unclassified from reading classified files, those users will be permitted to read the unclassified files containing the classified data. If temporary changes to weaker policies are to be supported, a solution must be developed to identify and sanitize contaminated files.

1.2 Document Overview

The report is structured as follows:

- Section 1, **Introduction**, defines the scope and this overview of the document.
- Section 2, **DTOS Overview**, reviews the DTOS system and its approach to security.
- Section 3, **DTOS Updates**, describes updates to the DTOS design and its security policy to address the requirements of adaptive policies listed in Section 1.1.
- Section 4, **Implementation Using DTOS**, discusses the effort to implement adaptive policies using DTOS.
- Section 5, **Assess Feasibility**, assesses the feasibility of implementing various approaches on the DTOS architecture.
- Section 6, **Fault Tolerance and Security**, examines how fault tolerance and security interact.
- Section 7, **Conclusion**, summarizes the results of this work and discusses open issues.

Section 2

DTOS Overview

DTOS was designed around a security architecture that separates enforcement from the definition of the policy that is enforced. This architecture allows the system security policy to be changed without altering the enforcement mechanisms. The policy is defined as a function from the security context of the subject making an access and the security context of the object being accessed to a set of permissions. Enforcement consists of determining whether the permissions specified by the policy are adequate for an access being attempted. The generality of the DTOS security architecture has been studied as part of the DTOS program [20]. The conclusion of this study is that a large variety of security policies, useful for both military and commercial systems, can be implemented.

The basic DTOS design is a microkernel, which implements several primitive object types and provides InterProcess Communication (IPC), and a collection of servers which provide various operating system services such as files, authentication, and a user interface [5, 13]. Of particular interest is a *Security Server* that defines the policy enforced by the microkernel and also possibly by other servers. When a request is made for a service provided by the microkernel, the microkernel sends identifiers for the security contexts of the subject and of the object to the Security Server. A context contains attributes about a subject or object that are necessary for making security decisions (since the information that makes up the context is dependent on the policy, the actual contexts are local to the Security Server and are not available to the microkernel). The Security Server then computes permissions for the context pair, as defined by the policy that it represents, and replies to the microkernel. Finally, the microkernel determines if the permissions required for the request were present in the reply. Other servers can communicate with the Security Server in a similar fashion.

For example, a Security Server implementing an MLS policy might maintain subject and object contexts consisting of a level and grant a write permission if the level for the object security identifier dominates that of the level for the subject security identifier and read permission if the level for the subject identifier dominates that for the object identifier (both permissions are granted if the levels are equal). A file server would check for write permission before allowing a request to alter a file. Conversely, a Unix-style Security Server might maintain a user and a group for each subject context and an owner, group, and access control bits for each object context and grant permissions from the access control bits depending on whether the user in the subject context matches that of the owner and whether the groups match.

A prototype DTOS microkernel and Security Server has been built by Secure Computing. The microkernel is based on Mach, developed at Carnegie Mellon University [12, 17]. The object types implemented by the microkernel include *task*, *thread*, and *port*.

Tasks and threads represent the active subjects, or processes, in the system. Each task has a security context that is used for security decisions involving that task. The state of each task includes a virtual memory consisting of a set of disjoint memory regions, each of which is backed by a server that is used to swap pages of the region in and out of physical memory. Each task contains a collection of threads, each of which is a sequential execution, that share the task's virtual memory and other resources. A server is implemented as one or more tasks.

The ports are unidirectional communication channels that the tasks use to pass messages. Tasks use *capabilities*, kept in a task name space, to name ports. Each capability specifies the right to either receive from or send to a particular port. These capabilities may be sent to another task in a message. For each port, there is exactly one receive capability and therefore at most one task can receive from the port (no task is able to receive from a port for which the receive capability is in transit rather than in a name space). IPC is asynchronous in that messages are queued in the port and the sending task does not wait until its message has been received (an exception is when the microkernel is the receiving task, in which case the sender waits until the microkernel finishes processing the message).

Sending or receiving a message is a Mach microkernel operation to which DTOS has added security controls that enforce the security policy. Thus, possession of the appropriate capability for a port is necessary but not sufficient in order to send or receive a message from that port. The security contexts of the task and the port must also permit the operation. The policy also constrains what capabilities may be passed in a message sent or received by a task.

The Security Server receives requests from the microkernel through the *microkernel security port* and from other servers through a *general security port*. Requests contain an operation identifier (allowing the Security Server to specify history-based policies that depend on the sequence of operations made on an object), a subject security identifier SSI (representing the security context of the subject), an object security identifier OSI (representing the security context of the object), and a send capability for a reply port. The Security Server replies by sending the permissions for that pair to the reply port (Figure 2-1). Not shown in this figure is the fact that the Security Server both defines and

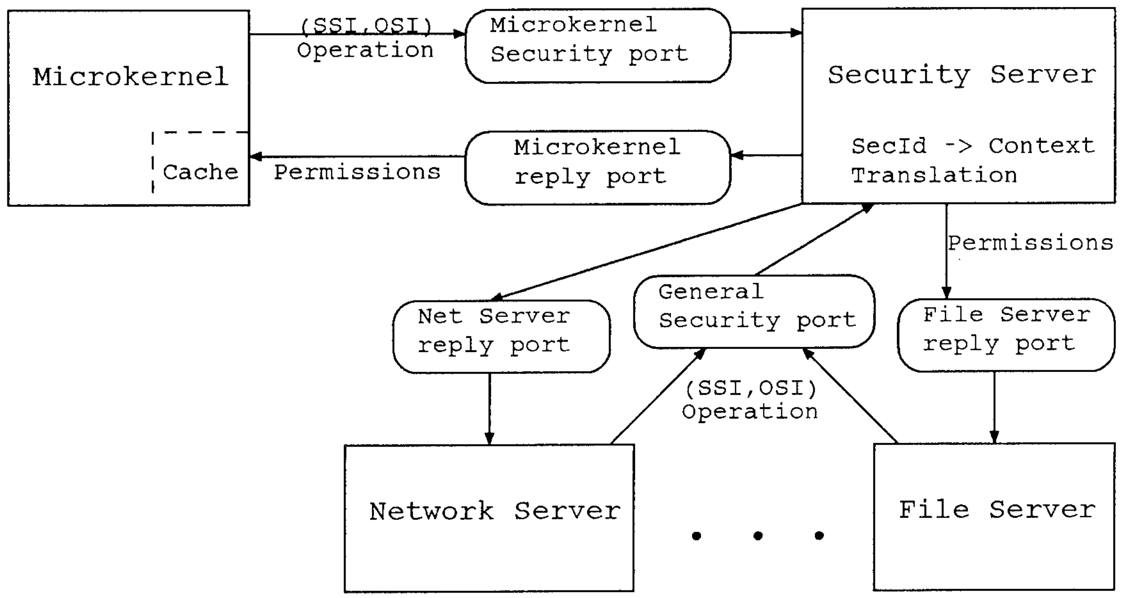

Figure 2-1: Security Server Interaction

enforces a policy for the requests that it receives. It might allow security determination requests from some subjects, but not from others. Similarly, it might allow security determination requests from a particular subject only for certain (SSI,OSI) pairs.

Security enforcement as described above would be very expensive due to the large number of messages that must be exchanged between the microkernel and the Security Server. The solution in DTOS is to cache (SSI,OSI) pairs with their permissions in the microkernel [13]. When the microkernel receives a request, it first looks in the cache for the appropriate (SSI,OSI) pair. If that pair is in the cache, the microkernel uses the cached entries. Otherwise, it sends the pair to the Security Server to determine the permissions, usually also caching the reply (part of the permission set returned is permission to cache the reply — caching is not permitted for permissions granted for a single operation by a dynamic policy). Since sending to and receiving from a port are microkernel operations controlled by the policy, the cache must be preloaded with permission for the Security Server to send and receive from the designated ports.

The DTOS prototype is available for use in studying secure systems. This availability and the separation between policy and enforcement made it attractive for studying adaptive security. The work described in this report discusses refinements to the design that are important for these policies.

Section 3
DTOS Updates

We studied the DTOS design and the security policy with respect to requirements of Adaptive Security Policies described in Section 1.1. We then specified updates to the design and the policy that would correct any shortcomings that we found. The particular requirements addressed were:

- the ways that a policy change can occur and how those changes are controlled,
- the places where security policy information is cached and the ways that these caches can be controlled, and
- auditing of system actions during times of relaxed security.

Proposed changes have been discussed with the DTOS team and modifications to the DTOS prototype and documentation have been scheduled where appropriate.

The designs for the DTOS microkernel and the initial Security Server, which supports an MLS policy enhanced with Type Enforcement [2, 16], were studied. These designs were as specified in the Formal Security Policy Model (FSPM) [19] and the Kernel Interfaces Document (KID) [21] and supplemented where necessary by looking at the implementation. Any errors or inconsistencies found during this study wêre reported to the DTOS team.

3.1 Changing Policies

Three methods were examined for changing the security policy enforced by DTOS:

1. Several policies are incorporated into a single Security Server and an interface to switch between policies is provided.
2. The current Security Server passes the receive capability for its security port to another Security Server that implements the new policy.

3. Some external agent designates a new security port that is received from by a different Security Server.

The first two methods were tested using the prototype, as described in Section 4. While the third is implemented in DTOS, it is not as useful for adaptive policies as the first two.

The first method is appropriate for table-driven policies, such as those supported by the initial DTOS Security Server. This Security Server determines permissions for a subject to an object by using the security contexts of the subject and the object as indices into the table. The selected entry contains the permitted accesses. Thus, the table specifies the policy and changing to a new table changes the policy. The Security Server provides a command **SSI_load_security_policy** that is used to replace the table (the initial DTOS Security Server receives this command on its general security port). The ability to replace the table is restricted to those subjects with *ss_gen_load_policy* permission in the current security policy.

This method is also useful if the translation between security identifiers and contexts is table driven. The context of a group of subjects sharing a security identifier can be changed by reloading this table. This mechanism seems to be especially useful when an MLS policy adapts by combining levels as discussed in Section 5. Unfortunately, the initial DTOS Security Server that we used does not use a table to make this translation (such a table has been added to the latest version), so we did not implement and test changing tables that define the security identifier to context mapping.

The second method, passing the receive capability for a security port, also requires the cooperation of the current Security Server. A Security Server provides a command **SSI_transfer_security_ports** (not originally provided by the initial DTOS Security Server) on its general security port, to which it replies with the receive capability for that port. This command is controlled by the current security policy in the same way that the **SSI_load_security_policy** command is controlled: only subjects with *ss_gen_transfer_policy* permission may issue this command. The initial DTOS Security Server design was extended to recognize this request and to return receive capabilities for both security ports to the handoff port specified in the request (Figure 3-2).

Transferring capabilities on DTOS from one task to another entails several security checks. While the receive capability for the microkernel security port is in transit, no Security Server can receive requests from the microkernel during this transfer and the permissions required must be cached in the microkernel. For the old Security Server these include: *av_can_send* and *av_transfer_right* to

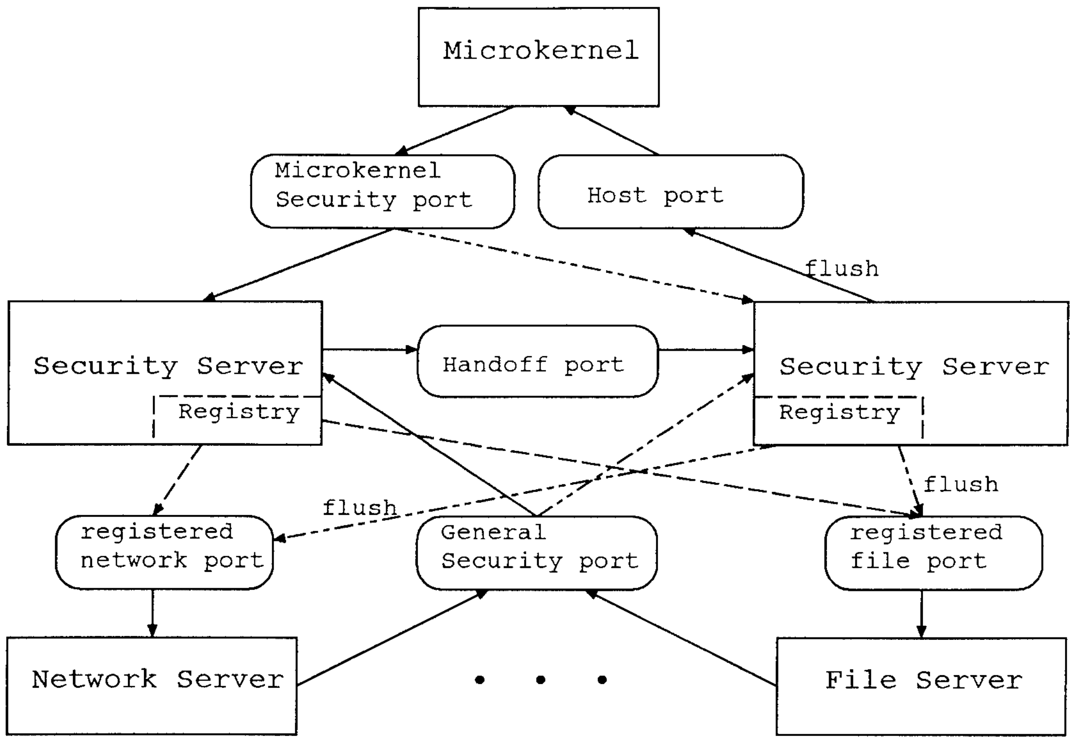

Figure 3-2: Security Server Handoff

the handoff port through which the transfer occurs, and *av_transfer_receive* to the microkernel security port being transferred. For the new Security Server these include: *av_can_receive* to the handoff port, and *av_hold_receive* and *av_can_receive* to the microkernel security port.

For the DTOS design, the only permissions that can be guaranteed to be cached are those wired into the cache when the system is booted. Thus, the wiring of Security Server permissions cannot be done if a Security Server can have an arbitrary security context, and we assume that there is some fixed context in which every Security Server operates. A more flexible design would be to add an **avc_wire_cache** command sent to the microkernel host port. This command could be used by both the old and the new Security Server before the handoff to ensure that the required permissions are in the cache (and possibly afterward to again make these cache locations available). Since this command was not necessary to test the handoff mechanism, we did not implement and test it (this command is on the list of possible DTOS enhancements).

The third method, designating a new security port, is needed for booting the system and for recovering from a Security Server failure, although it can also be used to switch Security Servers. A client of the Security Server, such as the DTOS microkernel, is notified of the designated security port through a **sec_set_security_server** command, which requires *set_security_server* permission. Of course, if this command is to be used in a catastrophic situation, this permission must be wired into the cache. This method provides for no coordination between the old and the new policies.

If a client can have several simultaneous outstanding Security Server requests, a policy switch can result in a sequentialization problem, caused when an access determination according to the old policy is received after access determinations according to the new policy. Thus, operations permitted by the new policy might be disallowed by the old policy after state changes permitted by the new policy have occurred. To alleviate this problem, policy determinations are tagged with a policy sequence number. If a client receives an access determination with a lower policy number than a previously received determination, it can discard the later determination and request it again. With the first method for changing the security policy, the Security Server increments the policy number when it executes a **SSI_load_security_policy** command. With the second method, the old Security Server increments the policy number when it executes a **SSI_transfer_security_ports** command and then transfers this number to the new Security Server along with the receive capability. With the third method, the policy number must be managed and assigned by the agent that sets the security port.

All three methods have been implemented in the DTOS system (the **SSI_transfer_security_ports** was implemented as part of this work). Experiments have been run utilizing the first two, providing a proof-of-concept for them. In particular, we have not found any deadlocks or other problems that occur during a policy change. The old and the new policies tested have been similar — each is a combination of MLS and Type Enforcement, as found in the initial DTOS Security Server.

On a distributed system, the Security Server would be distributed with a local representative on each node. This architecture provides the additional problem of maintaining coherence of the policy during a policy change. One solution is to maintain policy data in distributed virtual memory, for which coherence has been widely studied. We therefore would not expect any difficulties, but since the DTOS prototype operates on a single server we were not able to run any tests.

3.2 Cache Control

For efficiency reasons, system components that enforce a policy may cache policy information. Such caching is done by the DTOS microkernel. Also, most memory systems cache permissions for access to allocated pages of memory in page table protection bits. When the policy changes, cached security information that becomes incorrect must be invalidated. The DTOS microkernel design contains mechanisms for invalidating its cache, but not the page table protection bits.

The cache correctness problem can be handled by clearing the cache when the policy changes. In the DTOS design, the **avc_flush_cache** command, which requires *flush_permission* permission, allows the Security Server to signal the microkernel that the policy has changed and that any nonwired cached information may be wrong (any permissions wired into the cache must be correct for all possible policies). A Security Server sends this command to the microkernel host port as part of servicing a **SSI_load_security_policy** command and after receiving a reply to a **SSI_transfer_security_ports** command (the new Security Server requests that the cache be flushed before it starts to accept requests for policy determinations). However, this solution does not deal with caches that the Security Server may not know about, such as one maintained by the file server.

One solution is to allow service providers that cache access information to register a cache-control port with the Security Server (Figure 3-2). Then, the **avc_flush_cache** command can be sent to all of the registered service providers. Since the service providers are trusted to properly enforce the cached accesses, they can also be trusted to register and flush their cache when instructed to do so. We therefore added a **SSI_register_caching_server** command, which requires *ss_gen_register_port* permission, to the Security Server interface. The **SSI_transfer_security_ports** command transfers the registered cache-control ports to the new Security Server, along with the ability to receive from the security ports and the policy sequence number.

A service provider that switches policies by changing the port on which security requests are sent (for example, following a **sec_set_security_server** command) should flush its cache and register a cache-control port with the (potentially) new Security Server. To avoid confusion, this port should be different from the one registered with the old Security Server, which may still be active and capable of sending **avc_flush_cache** commands. Unfortunately, the DTOS microkernel does not explicitly register a port, but instead uses the host port for cache control. We could change this convention, but since we recommend the **SSI_transfer_security_ports** command for switching to a new Security Server and the use of the **sec_set_security_server** command only in exceptional circumstances, we have not.

Cache flushing that occurs during a policy change must be synchronized with outstanding Security Server requests. All Security Server requests from the microkernel or other servers are sent to the same port and are therefore processed in the same order that they are sent. Thus, all security requests from a server for which the replies represent a new policy must have been submitted after all of those for which the replies represent the old policy.

A problem exists, though, in that replies with old policy information might arrive after a flush command or after new policy replies. We stipulate that any old policy information must not be cached after the flush occurs. One solution is to discard any replies from Security Server requests that were pending at the time of a flush and to reissue these requests. With the microkernel, this solution was not needed. An old policy reply arriving after a new policy reply is equivalent to the case where the old policy reply arrives, the requested operation is allowed, the thread making this request is interrupted, the new policy reply arrives, and the thread waiting for this reply proceeds. The only way to avoid such a situation is to handle one request at a time or to restart all requests that the server is working on when a flush is received. We chose not to restart the microkernel operations when it receives a flush.

An alternative to flushing the cache en masse in response to some event is to incrementally flush it by timing out cache entries. DTOS allows the Security Server to specify a time bound for caching of permissions. A problem with this solution is that inconsistencies might arise because the cache contains values from an old security policy that have not yet expired along with values from a new policy. However, such inconsistencies might be acceptable in some cases. For example, suppose each entry in the cache is assigned a validity duration of 1 minute and the system policy changes in response to a declaration of war. Given the communication delays inherent between the decision to go to war and the receipt of the message, the fact that there is a 1 minute delay in the implementation of the new policy might be acceptable. This is analogous to the distinction between soft and hard real-time systems. Sometimes an immediate change in the policy is required, while other times a transition period is acceptable.

There are two places in DTOS that caching occurs outside of the explicit permission caches. The first is that a capability in a process list represents permission for that process to have the name of the object represented by the capability (but not necessarily to use that capability). This raises the issue of whether a secure system should control who can name an object, or just what can be done with that name. In DTOS, the exchange of names between processes, but not the holding of names, can be controlled by the security policy. Therefore, a change in policy cannot negate permission to continue to hold a name.

The second place, which is somewhat harder to deal with, involves the caching of permissions for access to memory objects in the virtual memory structure. The permissions for a DTOS task to a region of virtual memory depend partly on the protection requested for that task and the permission granted by the security policy. DTOS checks the security policy whenever the protection is changed using the Mach **vm_protect** command, but it should also be checked whenever the policy changes (as signaled by a cache flush or timeout). Unfortunately, the subject and object security identifiers used to index the cache are not easily mapped to the set of task and memory region pairs to which they correspond.

For history-based security policies in which individual subject and object security identifiers are flushed, the problem is especially severe. One approach is to keep a linked list of tasks with the same subject security identifier and a list of object security identifiers for the arguments in Mach **vm_map** commands. Whenever an object security identifier in this latter list is flushed, the security permissions for all memory regions of tasks in the linked list for the subject security identifier would be recomputed. However, since the focus of our work is on adaptive policies rather than history-based policies, we have not explored this further.

When switching to a new security policy, in which the entire cache is flushed rather than just several selected entries, permissions must be recomputed for all memory regions. We added this feature to DTOS by iterating through the tasks and, for each one, through the memory regions (this is on the list of possible enhancements for the distributed version of DTOS). The protection for the region is set to be the intersection of the current protection with the new permissions.

3.3 Security Identifier to Context Mapping

Security decisions in DTOS are based on identifiers representing the security attributes (the security context) of the subject attempting an access and the object being accessed. Different policies might be based on different attributes and therefore some policies may treat two subjects or two objects differently while other policies treat them the same. Thus, subjects that are treated differently in any of the possible policies must have different identifiers and policies that are coarse in their differentiation among subjects must be able to treat several different identifiers as representing the same context.

The initial DTOS Security Server treats a segment of a security identifier (referred to as the MID, or mandatory identifier) as an encoding of security level and type enforcement attributes. A policy with coarser levels can be implemented by changing the test for strict equality among levels to one that tests for relatedness. Implementing a more fine-grained policy, or one based on other attributes, requires use of the remainder of the security identifier (referred to as the AID, or authentication identifier) to differentiate among subjects and objects that have the same level and type class.

3.4 Auditing

An audit log can provide important information to be used during recovery from a policy change. If the system records accesses that are permitted during a period in which the policy is weakened, this record can be analyzed to identify information flows that violate a stronger policy. Steps can then be taken to isolate this information.

Only security-relevant events should be audited. Auditing all system accesses is likely to require a large amount of storage space and processing time. Also, analysis of such a large volume of information is likely to be expensive. Since what is 'security relevant' will vary with the security policy, determination of what events to audit should be made by the Security Server. However, the Security Server cannot itself do the auditing because the microkernel cache intercepts many of the access checks. Thus, the microkernel must have a mechanism by which it can be notified by the Security Server of those events that should be audited.

In this project, we only addressed the problem of determining how policies changed and what should be audited in a constrained context (Section 5.1.2). Determining this information automatically would be impossible if the policies can change in arbitrary ways. In situations where there are a small number policies with a predetermined set of transitions, auditing information can be built into a Security Server. In other situations, this might have to be an input from outside the system.

The DTOS design does not include any facilities for auditing. However, a mechanism in the design (that had not yet been fully implemented) provides much of the required microkernel capabilities. When the Security Server supplies a list of allowed accesses for a subject security context to an object security context, it also supplies a list of accesses for which it should be notified. While this facility was originally intended for use with dynamic security policies, it can be used to instruct the microkernel to notify the Security Server about security-relevant events. The Security Server is then able to audit the events about which it is notified.

A couple of extensions to the notification mechanism are useful. The first is a microkernel switch that allows the Security Server to specify that all security failures should be reported. If the switch is set (this should be the default case), the microkernel will notify the Security Server about any accesses that are attempted in violation of the security policy. The second extension is to designate a special port for notifications so that notifications are separated from requests for security decisions. The ability to receive from this special port can also be passed to a special Audit Server.

3.5 Conclusion

Adaptive security policies can be implemented on DTOS with only a few modifications to the microkernel:

- It must recompute page table permissions when its permission cache is flushed.
- It must provide an audit port to which it sends a notification of security-relevant events.
- It must provide a means for specifying which events should be audited.
- It should provide a means for dynamically specifying which entries in the permission cache must be wired.

As anticipated, the most significant changes required for adaptive policies are for the Security Server:

- It must specify which events should be audited.
- It must provide a mechanism for registering servers for which it will provide security information.
- It can:
  - provide a mechanism for handing off capabilities to another Security Server for the security ports and from the registry,
  - provide a mechanism by which the mapping from security IDs to contexts can be changed.

Section 4

Implementation Using DTOS

Several changes were made to the DTOS prototype and its initial Security Server, primarily corresponding to the DTOS analysis described in Section 3. These changes were tested and, where appropriate, the performance was measured.

4.1 Audit

Auditing required modifying the microkernel and the Security Server. We also added an 'Audit Server' that interacts with the file system to make the audit log available for study (an actual server would also analyze and filter the log, but this was outside the scope of the current project). Finally, we measured the impact of our changes on system performance and studied a log for sufficiency of information.

The information supplied by the Security Server to the microkernel includes a *notify vector* that specifies the events for which the microkernel should send a notification to the Security Server. This vector, kept in the cache with the permission vector for a (SSI,OSI) pair, has a set of flags corresponding to the flags in the permission vector for that pair. Whenever a permission is checked, the microkernel also checks the notify vector and if the flag for that permission is set, a notification is sent to the security port. We modified the microkernel so that the notification is sent to a separate audit port so that a server other than the Security Server can receive notifications, that notifications contain the information needed for an audit, and that notifications are buffered to reduce communication overhead.

The first microkernel change was to add a request that sets the port to which audit information is sent. This was done by combining the request with the one that sets the security port to form a new **host_set_special_port** request. A client needs a send capability to the microkernel host port and *set_special_port* permission for the microkernel host security context to make this call and *set_audit_port* permission to set the audit port.

The next microkernel change was to create an audit buffer. The macro that is used to make a permission check was modified so that it also checks the notify vector and, if the notify flag for the permission being checked is set and there is a valid audit port, adds a block of information to this buffer. If adding the block causes the buffer to become full, a full-buffer signal is raised, signaling that the buffer should be sent.

The last microkernel change was to add two microkernel threads. The first blocks until a timeout occurs (once each second, provided that the buffer is not empty) and then sends a full-buffer signal; this ensures that audit information is regularly supplied to the system even if audited events rarely occur. The other thread waits for a full-buffer signal, generated when either information is added to the buffer or a timeout occurs, sends the buffer to the audit port, and then waits again. Note that if there is no audit port, nothing will be added to the audit buffer, the full-buffer signal will not be sent, and this second thread will remain blocked.

Making the Security Server be the receiver for the audit port, possibly by setting the audit port to be the microkernel security port, allows for most of the functionality of original notification design. However, the buffering destroys the synchronization between notification events and permission checks. If this synchronization is important, notification can be accomplished by not allowing the appropriate permissions to be cached.

The change to the Security Server consisted of adding the audit vector to its database. Originally, the Security Server would always send a null notify vector. This was replaced by code that got a vector from the database, along with the permission vector. We then modified the database to include these vectors, most of which were set to no auditing. The exception was for those OSIs that specify task ports, for which all memory permissions (such as *allocate_vm_region* and *deallocate_vm_region*) are set to audit.

The information audited initially consisted of the SSI and OSI that specify the permission vector, the permission vector, and the permission being checked. While this allowed us to determine what accesses were made by a class of subjects to a class of objects, tracing the sequence of accesses made by a particular subject to a particular object was impossible. We therefore added identifiers for the task and the thread of the subject.

The usefulness of the task and thread identifiers are due to the way that requests sent to the DTOS microkernel are implemented. Instead of the request message being added to a port and later received by a microkernel thread, the client thread traps and continues in the microkernel address space. Thus, the task and thread identifiers for events performed within the microkernel are those of the client. The same situation does not apply to requests sent to other servers. A useful extension would be to add a message identifier to send and receive events. We could then link the client thread that sends a message with the server thread that receives it. Calls by the server to other servers could also be linked to the client.

Several problems were uncovered while testing the changes. The first was that if the Audit Server (the task with the receive capability for the audit port) died, the microkernel would continue to send buffers to the audit port, filling up memory. The solution was to add code so that the microkernel tests for a dead port before sending a buffer to it. If the port is dead, auditing is turned off. The second problem was that the new thread overflowed its stack due to the size of the audit buffer. The solution was to give it a larger stack than any of the other microkernel threads. The third problem was that the microkernel deadlocked when it waited until the audit buffer became full to send it, since sending it may generate additional audit events. The solution was to trigger the new thread before the buffer becomes full and to run this thread at a high priority.

We ran a performance check on auditing, using as a benchmark the compile of the IPC part of the DTOS microkernel. The results were as follows:

Cost of microkernel audit code. Auditing requires each time a request is made that the microkernel check whether or not the request should be audited. We ran our tests without the audit checks compiled in and then with the audit checks in the code (but all auditing turned off). The result was an increase of only .12% in the execution time.

Cost of auditing VM checks. When we turned on auditing of virtual memory checks, the run time of our tests increased by 10.8%. Our tests showed that this is primarily due to processing by the Audit Server, including the time required to write the information to a file (although some overhead is involved in context switches and by the microkernel recording audit information). Part of the cost of the Audit Server is that the number of permission checks performed by the microkernel went up 23%, from 279,550 to 344,560. The number of events audited was 125,324, about 36% of the permission checks.

Based on these results, the cost of the audit mechanism is acceptable. When auditing is not used, the impact is negligible. The amount that it is used can be specified as part of the security policy and while the effect of auditing a substantial number of events is noticeable on the system performance, auditing does not overwhelm the other processing on the system.

4.2 Handoff

We extended the Security Server so that it could pass control to another Security Server. This extension included two new commands, code to carry out the handoff, a server registry, and a policy identifier, as described in Section 3.1. We then tested a handoff between two Security Servers, each implementing MLS with Type Enforcement. During this work, we changed the way that the DTOS microkernel handles outstanding permission checks, which had the side effect of fixing a sequencing problem which could occur if two permission checks for the same (SSI,OSI) pair occur concurrently. A problem uncovered was that extra security permissions are required when a Security Server is started up from Unix than when it is started during system startup.

The new Security Server commands are **SSI_transfer_security_ports** and **SSI_register_caching_server**, requiring *ss_gen_transfer_policy* and *ss_gen_register_port* permissions respectively. These commands are issued through the general security port. The **SSI_register_caching_server** command provides a send capability to a port, which the Security Server stores in its registry. For testing purposes, we created a special Test server that registered with the Security Server and then monitored interactions with the Security Servers.

The **SSI_transfer_security_ports** command provides a send capability to the port through which the handoff will occur, for which the new Security Server is the receiver. This command could potentially be initiated by a monitor subject, rather than the new Security Server, that determines that a special situation has occurred that requires the policy to adapt. The security identifier for the handoff port must be such that send and receive permission for a Security Server security identifier are wired into microkernel cache when the system is initialized. The Security Server receiving the command increments its policy counter and sends the receive capabilities for both security ports, the capabilities contained in the registry, and the policy counter to the handoff port. Any commands queued in the general security port at the time the **SSI_transfer_security_ports** command is issued will be processed before the handoff occurs, since the **SSI_transfer_security_ports** command also queued in this port. Any commands sent to the microkernel security port before the **SSI_transfer_security_ports** command is received may or may not be processed before the handoff occurs.

The new Security Server initially reads a policy database from the Unix file system and then waits for a message from the handoff port. When a message arrives, it initializes its registry with the capabilities from the old registry and uses each to send a flush command to the registered servers. It then sends an **avc_flush_cache** command to the microkernel host port (since messages sent to the microkernel are synchronous, the Security Server will be delayed until the microkernel receives and responds to this message). Finally, it begins service by waiting for messages from the security ports.

We looked at the synchronization of policy changes with requests from the microkernel for policy decisions. In particular, Security Server replies issued before a flush must not be cached after the flush. This is only a problem if the reply is received after the flush. The solution adopted is to allocate a cache cell at the time that a request is made to the Security Server and to mark that cell as busy. When the flush occurs, all allocated cache cells that are not wired are moved to the free list, but the cell for the pending security check will remain busy. A cell on the free list is not reallocated until its busy flag is cleared; the value returned by the check will be used as the permission for the request, but will not be cached since the cell into which it is written is on the free list. This busy flag solution also solves a synchronization problem with concurrent checks for the same identifier pair. Previously, concurrent checks would have each generated requests to the Security Server with the possibility that the reply to the second could arrive before the reply to the first, resulting in a problem with history-based policies where the sequence of checks is important. With the busy flag, the second request will delay until the busy flag is cleared before potentially generating another call to the Security Server.

The solution given here might not work for some other servers, or those servers might have stricter synchronization requirements. We therefore added the policy identifiers to replies from the Security Server. This allows a server to discard 'old' replies, especially those received after a flush, or to delay action on 'new' replies received before a flush (since the flush command with these servers is not synchronous, replies based on a new policy can arrive before the flush command for the old policy).

This work was more complicated than we had originally anticipated. These problems were the result of interactions with UNIX during the transfer, as opposed to starting up a server from the boot process on a bare machine. The first problem occurred when we tried to load the Security Server, which had been linked with the Mach linker to the Mach libraries, under the Lites operating system; the environment that Lites establishes was incompatible with the Mach library. After relinking the Security Server, we discovered that the security policy did not contain the necessary permissions for a Security Server to communicate with Lites or for Lites to control a task with the Security Server security context. For example, Lites could not destroy the Security Server task after control had been handed back to the original Security Server. We therefore had to add some new permissions to the policy.

Security Server handoff has been tested, including a handoff from one server to a new server and back again, and appears to work as designed. We could have measured the overhead of a handoff, but since this overhead is an absolute measure rather than a comparative measure like those that we presented for auditing, we were unsure of the significance of any of the measurements. For example, the choice of processor and the operating system on top of DTOS would have affected our results. Also, the actual handoff is only part of the processing that occurs in order to produce a policy change. The new Security Server might need to be started and its tables loaded, which depends on the response time of the system to the signal to change policies and the size of the tables. We therefore did not make these measurements, although on our implementation the time required for the handoff was under a second.

4.3 Page Table Reset

We added a mechanism to the DTOS microkernel to reset the page table protections when the permission cache is flushed. DTOS calculates the protections at the time that a memory region is created for a task, using the intersection of the protections requested and those allowed by the policy. Also, the protections are recalculated whenever a **vm_protect** request is received specifying different protections for a region. This code queries the policy and intersects the requested protection with those granted by the policy and then sets the page table protections. To recalculate protections when the policy changes, we added code that cycles through each memory region of a task, essentially doing a **vm_protect** that specifies the current protection. This code is called for each task with a security identifier affected by a flush.

One glitch that we discovered is that the call by the Security Server to flush the cache is synchronous. Therefore, if the microkernel tries to reset the page table permissions during the call, the Security Server will be blocked and unable to supply the necessary permissions. The solution was for the microkernel to return from the call immediatedly after starting up a new thread to reset the page tables.

Concerns about a potential deadlock while the permissions for the Security Server memory are recomputed turned out to be unfounded, since the old page table protections remain in effect until the new ones are calculated. Thus, assuming that the Security Server had permission to execute its code before the flush command (and that this permission was set in the page table when the Security Server's memory was mapped to the code), it would retain this permission in the page table until the new permission for that region was computed. However, security checks other than those needed to recompute memory protections must be postponed until the memory protections are complete. Otherwise, a task could use permissions from the new policy to read from file A and use permissions from the old policy to write to a memory area backed by file B, even though neither policy permits the task to transfer from A to B.

Section 5 Assess Feasibility

The goal of this task was to look at the feasibility of implementing some of the approaches outlined in the Odyssey Research Associates, Inc. (ORA) final report for Rome Laboratory Contract F30602-94-C-0111 [9] on the DTOS architecture. In particular, we looked at dynamic security lattices and at task-based access control. We also looked at delegation policies, which can be used to implement application firewalls. In this section, we show how each of these approaches can be handled.

5.1 Dynamic Security Lattices

The dynamic security lattice model is a generalization of the security lattice model normally used to describe transitive security policies such as MultiLevel Security (MLS). A security lattice consists of a set of security levels that are partially ordered by a *dominates* relation. The dominates relation is transitive (if level l_0 dominates l_1 and l_1 dominates l_2 then l_0 dominates l_2), antisymmetric (if level l_0 dominates l_1 and l_1 dominates l_0 then l_0 equals l_1), and reflexive (level l_0 dominates l_0). A dynamic security lattice allows several security lattices to be represented within a single model and can therefore be used to represent a class of adaptive security policies. In the following discussion, three different dynamic security lattice models are presented. They are related in that each models system entities and the information flows that are allowed.

5.1.1 Background

This section gives some background information on dynamic security lattices, based on papers by ORA and by Badger [9, 1].

5.1.1.1 Dynamic Security Lattices - ORA Report. In the ORA report, dynamic security lattices define an *effective* security lattice, where the dominates relation is determined by the transitive closure of the information flows that are enabled. The nodes of the lattice represent potentially different security levels. Nodes can be collapsed together to enable information flows not allowed by the original lattice. This leads to an effective lattice of security levels that is coarser than the original lattice. The primary advantage of a dynamic security lattice is that it provides a rough estimate of where sensitive information may have leaked following a change in security constraints.

In the ORA report, there is a *current security lattice* that is currently in effect, as well as an *official mandatory security lattice* with which the current security lattice must not disagree. The current security lattice must always be compatible with the official mandatory security lattice in the following ways:

- If a user is forbidden under any circumstances to learn information, the user's level can never dominate the level of the information.
- When information is exported from the system, the information must have a level that dominates the levels of the sources of the information (unless there is an official downgrading policy).

Typically, the above constraints imply that all users must be cleared for all information that they could ever receive. Dynamic security lattices in this case are a way of enforcing a need-to-know policy.

To help explain the impact of dynamic lattices, the ORA report attempts to formalize the issues. They assume a computer system (operating system plus applications software) can be described by the following parameters:

- A set E of possible information-containing entities. This set will include system variables, files, and users. The level of entity e is represented as $l(e)$.
- A set A of *actions*.
- For each action a, there is a set $R(a)$ of entities that are read by the action, and a set $W(a)$ of entities that are written by the action.
- At any time, there is a set P of permitted actions.
- The *effective level lattice* with dominates relation $\succeq$ is defined by the requirement that for any two entities u and v, $l(v) \succeq l(u)$ holds if for some sequence of entities $< e_0, e_1, ..., e_N >$ and actions $< a_0, a_1, ..., a_{N-1} >$ with $a_i \in P$:
  - $e_0 = u$
  - $\forall j < N. e_j \in R(a_j)$

- $\forall j < N . e_{j+1} \in W(a_j)$
- $e_N = v$

To create a policy for a system, the circumstances under which an action will be enabled, and a set of constraints (policy situation-dependent) on the flow relations are specified. In some circumstances, the granting of new information flow rights (enabling new actions) may need to be accompanied by the rescinding of other information flow rights in order to prevent the transitive closure of the flow rights from connecting two entities that should never be connected. For example, the mandatory security policy might prohibit information from ever flowing from A to C, although in certain circumstances the current security policy allows information to flow from A to B. To enforce such a policy, information flow from B to C is severed when information flow from A to B is enabled.

5.1.1.2 Dynamic Security Lattices - Badger Paper. The Badger paper [1] provides the foundation for the technical work presented in the ORA report [9]. In this paper, relaxation security is expressed in terms of the guarantees that a trusted system may provide about the manner in which information has been and will be allowed to flow between subjects and objects. After a security relaxation, a trusted system is able to provide some smaller set of guarantees.

The Badger paper defines a *relaxation lattice* as a lattice A of automata which are identical in every way, except possibly for their state transition relations. The automata are parameterized by subsets of C where C is a set of security constraints (i.e., guarantees) which are defined as accesses prohibited for a subject to an object. An automaton A is less than automaton A' if its constraints are a subset of those of A'. Thus, the lattice is oriented such that the automaton which satisfies the largest set of constraints, and thus accepts the smallest language, is at the top.

The Badger paper also defines the concepts of *relaxation security* and *strong relaxation security*. During a period in which the set of guarantees that the system should provide does not change, a *relaxation secure system* prohibits the violation of guarantees that the system is still able to support. Intuitively, a relaxation secure system moves through a number of phases, providing a particular set of guarantees in each phase. In each phase, a relaxation secure system will prohibit those operations that would violate currently promised guarantees *which have not already been violated* in previous, more relaxed phases. This definition is motivated by the need to provide guarantees about what has *not* happened in a system which permits security relaxation.

In relaxation security, once a particular information flow has occurred, further information flow in the same manner is not prohibited. This definition of relaxation security is not appropriate for all applications. To constrain such behavior, the Badger paper introduces a set of *strong* constraints which must be honored regardless of past violations. *Strong relaxation security* is similar to relaxation security with the addition of these strong constraints. Thus, an operational system may at a particular time have a set of normal constraints and a set of strong constraints. The strong constraints must be always satisfied while the normal constraints are satisfied only if they have not been previously broken.

The ability to relax and reimpose security constraints defines a partial security recovery scenario: when constraints are reimposed, recovery occurs automatically to the extent that no damage occurred.

A more active recovery is required to enable the use of information that might have been mislabeled. If possibly mislabeled information has been exported to the external environment, recovery of the information is not possible, although the TCB may provide assistance concerning where and to whom information was disclosed. If possibly mislabeled information has not been exported, recovery to a set of constraints $C_i \subseteq C$ is achievable if every subject and object into which information flowed in violation of a constraint in C_i was checkpointed during system relaxation and if system integrity constraints will permit a rollback to the state of these objects before the period of constraint relaxation. In this case, recovery is accomplished by deletion of possibly mislabeled objects and substitution of the checkpointed versions. In those cases where a rollback is not feasible, manual review of mislabeled subjects and objects is necessary to reestablish security.

Note that in the ORA report the lattices denote information flow, while in the Badger paper the lattices denote security constraints. These two notions are related, however, in that security constraints imply the entities between which information can and cannot flow in a system.

5.1.1.3 Dynamic Security Lattices - A Simpler Model. While the descriptions above are adequate for describing MLS systems, they are overly complex and inflexible. We therefore developed the following simpler formalization of an adaptive security policy and the resulting dynamic lattice:

- A policy is a directed graph $< N, F >$ where N is the set of information-containing nodes and F is the set of directed edges that correspond to the allowed information flows.

- The level of a node n, $l(n)$, is isomorphic to the strongly-connected component containing n. That is, $l(n) \approx \{x | n \rightarrow^* x \wedge x \rightarrow^* n\}$ (a level is represented by the set of nodes such that information may flow from each node of the set to any other node of the set).
- The effective level lattice corresponds to the graph where each strongly-connected component is represented by a single node.

This description is simpler and uses existing mathematical concepts more directly. Hence, it is easier to reason about.

The ORA entities E correspond directly to the nodes of the simplified model, N.

An ORA action, a is a multi-edge that carries information from its read set $R(a)$ to its write set $W(a)$. In the simplified model, that action is modeled by a collection of allowed flows or edges. For each node x in $R(a)$ and each node y in $W(a)$, an edge (x, y) is added to F, the set of directed edges.

The ORA notion of permitted actions is not represented directly. At any time, the set of permitted flows in the simplified model is simply the set of edges, F. To compare two policies, you must specify two sets of edges.

Two nodes are in the same security level if they are in the same strongly-connected component. That is, if there exist information flows with zero or more edges from a to b and from b to a, a and b are at the same security level.

The lattice formed by reducing strongly-connected components to single nodes defines the effective level lattice. This reduced graph is a directed acyclic graph (DAG). Here is a place where we can leverage existing definitions and algorithms. The standard algorithm for finding strongly-connected components runs in time linear in the size of the graph.

The ORA and simplified models are not isomorphic. The simplified model does not enforce the clustering of information flows imposed by the ORA model's actions. If all the actions of the ORA model had a read set of size one and a write set of size one, the two models would be isomorphic.

5.1.2 Using a Dynamic Security Lattice

The ORA and Badger reports defined sets of related security policies, but failed to indicate how they might be used. For dynamic security lattices to be useful, we must be able to determine the security impact of operating under multiple policies. Again, the simpler model is easier to use and understand. In that model the only possible changes are to add or delete flow edges.

Start with an initial policy, add or delete some flow edges, and let some information containing messages flow over those edges. Now, what happened? Can we revert back to the original policy? If so, what needs to be done?

Clearly, deleting edges decreases the possible information flows. Nothing "bad" can happen with respect to the security defined by the initial policy. Any message sent under the revised policy could have been sent under the initial policy.

Adding an edge is more complicated.

If the new edge is in the transitive closure of the initial F, nothing essential has changed. The information in that message could have reached the same nodes using a different path. Note that this assessment depends on a particularly simple notion of security, ie. the source and destination of the allowed information flows. Security that depends on enforced paths is not addressed. For example, in a policy that requires all information reaching node C to pass through node B, adding an edge from A to C would subvert the policy. More complicated notions of security are addressed below.

If a message crosses a new edge not already in the transitive closure of F, we say the message is *dirty*. A dirty message is one that carries information from one node to another, but was not allowed by the initial policy.

Clearly, dirty messages must be audited. They will be needed when reverting from a more permissive to a more restrictive policy.

A node is *tainted* after it receives a dirty message. Any message leaving a tainted node is dirty since that message may contain information from a previous dirty message.

To enable the restoration of security after operation under a permissive policy, tainted nodes must be sanitized. One simple possibility is to "roll back" tainted nodes to their untainted state. Either of the standard database schemes, checkpoint-restore or log-undo, could be used. Roll back of tainted nodes is not a complete solution. Physical I/O cannot be undone. Once classified information is printed and dispersed, it is very hard to recover it. Another weakness of the roll back scheme is that restoring only the tainted nodes can break more global system invariants. For example, if two nodes, one tainted and the other not, count the messages they receive and by design must have equal counts, after the roll back the counts will differ.

5.1.3 Dynamic Security Lattice Implementation on the DTOS Architecture

We analyzed how the dynamic security lattice model can be accommodated in the DTOS architecture and determined enhancements needed to the DTOS architecture to implement this model. We assume that any security policies satisfy the information flow transitivity requirements of the ORA dynamic security lattice model (Section 5.1.1.1). In other words, if information should not flow between entities A and B, a security policy on the DTOS system must prevent information from flowing either directly between A and B or indirectly through intermediate nodes.

We will next discuss three possibilities for implementing dynamic security lattice model security policy changes on the DTOS architecture. These possibilities are changing Security Identifier (SID) assignments, changing the mapping from SID to security context, and changing the algorithm that decides which MLS access vector applies to a given situation.

5.1.3.1 Changing Security Identifier Assignments.

This section discusses implementing dynamic security lattice model security policy changes by changing Security IDentifier (SID) assignments. Since DTOS SIDs directly encode information including security level and category information, collapsing security levels for the dynamic security lattice model can be accomplished by assigning new SIDs to those entities in the system that have SIDs assigned to them. For example, if the levels *secret* and *top-secret* are to be collapsed to the single level *secret*, all entities in the system which currently have a *top-secret* classification can be assigned new SIDs which give them a *secret* classification.

Reassignment of SIDs in the DTOS system can be accomplished as follows. Entities that have SIDs assigned to them in the DTOS system are tasks, ports, and VM map entries. All tasks, ports, and VM map entries can be accessed by a task going through the appropriate data structures and changing the necessary SID assignments. All tasks can be accessed because Mach keeps a queue of all processor sets and each processor set keeps a queue of tasks assigned to it. Every task in the system can be accessed by traversing the queue of processor sets and the corresponding queues of tasks. All ports can be accessed through the task structures. Each task keeps a queue of threads it contains. All ports associated with a task can either be accessed through the thread structures associated with the task or through a table of ports which can be accessed through the task. All VM map entries can also be accessed through the task structures. Therefore, starting at the top-level processor set queue, all tasks, ports, and VM map entries in the system can be traversed and SIDs can be reassigned using a list of (FromSID, ToSID) pairs.

While changing SIDs may be fairly easy to implement, the ability to do so must be programmed into microkernel and also makes reverting to a previous policy difficult following the combination of several security contexts. Further, the SIDs for other ports, such as those that name the threads of a task, may be related to the SID of a task and therefore must be found and updated if the task's SID changes. Therefore, a Security Server design in which the SIDs and security contexts are tightly coupled, such as the initial DTOS Security Server, should not be used for systems that support dynamic security lattices (as well as other adaptive policies). We will not consider this solution further.

5.1.3.2 Changing the Mapping from Security Identifier to Security Context. This section discusses implementing dynamic security lattice model security policy changes by changing the mapping from SID to security context. Collapsing security levels in the dynamic security lattice model can then be accomplished by changing the mapping from SID to security context. For example, if the levels *secret* and *top-secret* are to be collapsed to *secret*, the SID to security context mapping would be changed for those entities that were at the level *top-secret* so that their security context would indicate that these entities are now at level *secret* instead of *top-secret*, thus effectively collapsing the *secret* and *top-secret* security levels.

Two possible implementations of the mapping are a table in the Security Server and a Context Server that performs the mapping. Unfortunately, neither was necessary for the policy specified by the initial DTOS Security Server and therefore was not included in it. With a table, a change in the mapping of SIDs to security contexts could then be accomplished by a command that signaled the Security Server to read in a new table or by switching to a new Security Server containing a different table. With a Context Server, a change in the mapping of SIDs to security contexts could then be accomplished by a command that switches to a new Context Server. A Context Server would probably require caching in the Security Server and was not used in our work.

5.1.3.3 Changing the Algorithm. This section discusses implementing dynamic security lattice model security policy changes by changing the algorithm that decides which MLS access vector applies to a given situation. Collapsing security levels in the dynamic security lattice model could then be accomplished by changing algorithms that affect the list of allowed accesses returned by the Security Server.

There are a number of possibilities for implementing an algorithm change in the DTOS system; four of these possibilities are:

1. Implement multiple algorithms in the Security Server code. A new Security Server command would signal a change and specify which algorithm should be used.
2. Send the executable for the algorithm to the Security Server when an algorithm change occurs. The Security Server would then map the executable to memory (the code would need to be relocatable) and jump to and execute the executable when necessary.
3. Send the address of a new algorithm executable to the Security Server when an algorithm change occurs. The Security Server would then jump to and execute the executable when necessary.
4. Specify the algorithm using some type of specification language. The Security Server currently gets domain, type, level, category, user, and access vector information from database files. This information is given via a specification language. A similar method could be used for changing an algorithm in that when the Security Server changes algorithms it looks in a specified file for the specifications for the new algorithm and then implements this algorithm change.

Depending on the method used to change the algorithm, implementing dynamic security lattice model security policy changes by changing the algorithm that decides which MLS access vector applies to a given situation could be fairly easy to implement. In addition, it seems that there would not be much of a performance hit to implement algorithm change using the four possibilities mentioned above.

5.1.4 Non-transitive Extensions to Dynamic Security Lattices

To model more complex security policies such as those built on type-enforced systems like DTOS, it is necessary to extend the simpler formalism described above. By attaching types to the messages and domains to the nodes, type enforcement eliminates the transitive behavior of simple MLS policies.

To model a type-enforced system, a subset of N is denoted as *trusted nodes*. Trusted nodes may modify or cleanse the information flowing through them. For example, a node that labels data before sending it to the printer is a trusted node. A node that filters top secret data before sending it on to a secret node is also a trusted node. These information flows are allowable in a type-enforced system and can be made non-bypassable, but they do not necessarily imply an MLS-like ordering of the nodes into levels.

The levels in a type-enforced system, to the extent that they exist, are computed by first deleting the edges leaving trusted nodes and then finding the strongly-connected components as before. This modification handles the case described above where adding an edge could bypass a required node. In a type-enforced system the node that must not be bypassed would be trusted. Hence, the edge leaving that node would not be included in the graph used to determine the levels. Adding an edge that bypassed the trusted node would detectably modify the allowed information flows.

The trusted nodes must be divided into two classes, robust and fragile. *Robust* nodes are trusted nodes that are trusted under any mutation of the security policy. For example, a dirty word filter can be trusted to always perform its job regardless of the allowed information flows. It simply removes the dirty words from any message it receives. *Fragile* nodes are trusted nodes under one policy, but are subject to spoofing or failure under changing policies. For example, a node that adds the "Secret" stamp to any pages passing through it to the printer is fragile. Under a modified policy that allowed top secret information to reach the labeler, top secret information would be printed with secret labels.

The rules used to guide auditing and recovery must be extended to handle type-enforced systems with adaptive policies. In a type-enforced system, messages leaving a robust node are never dirty. Even though the node may become tainted and require roll back, the messages that leave it are not dirty. Messages leaving tainted fragile nodes are dirty. The edges leaving a fragile node are not considered when computing levels, but the messages may be dirty.

5.2 Delegated Policies: A Specialized Form of Adaptive Policy

Occasionally it would be convenient to temporarily restrict a subject's permissions. For example, when viewing a PostScript document retrieved from the World-Wide Web, we would like to restrict the previewer's file access. The PostScript previewer is normally run by a trusted user on the internal network, inside the security perimeter. It is usually given permission to read and write files on the internal network. But, the PostScript programming language allows reading and writing files, so a piece of external data handed to a "trusted" internal process may wreak havoc. In this case we would like to tightly restrict which files the previewer can read or write. On systems where the security policy is flexible, to the point where vanilla users can modify it, we could simply create a new security context for the previewer when reading untrusted data. But, most secure systems do not allow for that level of flexibility. An alternative would be to temporarily delegate security decisions to another process. If done in a controlled way, that process could restrict a particular instance of the PostScript previewer.

Here is how it would work in DTOS. Rather than start the previewer directly, start its "delegated security manager" first. That manager process registers itself with the security server indicating its intent to manage the previewer. The security server returns a new SID for use by the previewer. Except for its identity, the context identified by that SID has the same properties as the normal security context assigned to the previewer. The manager starts the previewer with the new SID. Whenever the kernel asks the security server for a decision about the previewer, the security server forms an answer and forwards it to the manager. The manager modifies the answer and returns it the security server. The security server ensures that the manager did not increase the permissions and returns the answer to the kernel.

Even though we did not implement this temporary delegation extension to the security server, the advantages are clear:

- Unlike most secure systems which try to ensure that only trusted data reaches trusted processes, here trusted processes can be applied to untrusted data. The extension is *useful*.
- The delegated manager cannot override the mandatory policy enforced by the security server. The extension is *safe*.
- The full security server interface can be implemented by the delegated manager. Not only can it determine the permissions, but also the cache and audit characteristics of the managed process. The extension is *general*.
- The kernel and the managed process know nothing about the delegated manager. The extension is *localized* to the security server.
- Except for the managed process, the system performs as it did before. The extension is *efficient*.
- Very fine-grained, per-process decisions can be enforced without modifying the system defined mandatory policy. The delegated manager can use whatever scheme it would like to limit the permissions it grants. The extension is *flexible*.

To further improve this scheme, it should be the case that the delegated manager can, if it chooses, be the delegated manager for any subprocesses created by the process it is already managing. This additional functionality would require the manager to control any process creation requests on behalf of its managed process. We did not explore what changes would be required to DTOS to implement this enhanced functionality.

5.3 Task-based Access Control Policies

This section provides background information on task-based access control policies as stated in the ORA final report [9]. Note that the term *task* is used in this section, as it was in the ORA report, to refer to 'a piece of work' rather than a Mach process. *Task-based access control* policies make explicit the relationship between an agent's privileges and the tasks it must perform. Task-based access control emphasizes what needs to be done, rather than what the agent who will do it is called. In *role-based access control* policies, related responsibilities are usually grouped together to form a role. There is an implicit assumption that related responsibilities should be concentrated in one individual. This assumption might be wrong or inconvenient for several reasons:

- Giving one individual unconstrained privilege in an area may be dangerous. If several people must work together to play what might be conceptually considered a single role, they can keep each other honest.
- In a crisis, distributing privileges so that no person becomes a 'single point of failure' may be important.
- Because a task is typically more constrained in time and scope than a role is, determining the access rights that are needed to perform a particular task may be easier than determining the access rights that are needed to fulfill a particular role.
- Task-based access control provides more flexibility than role-based access control. An agent can be given the authority to do a task only once in a particular situation. In contrast, a role implies ongoing privileges.

In creating a task-based access control policy for an adaptive system, the following parameters must be defined:

- a collection of *scenarios* in which the system will have to operate. Each scenario would be a class of situations.

- the collection of *tasks* that may need to be performed for each scenario.
- the collection of *actions* that may need to be taken to accomplish each task.
- the collection of *objects* that are involved in each action, together with the type of access involved.

A well-known class of security policies that can be implemented using task-based access control is Clark and Wilson [4]. In these policies, an object is known as a Constrained Data Item (CDI) and an action is known as a Transformation Procedure (TP). These policies guarantee that a CDI can only be modified by a well-formed TP. The other aspect of these policies is *separation of duty* in which different tasks are executed by different processes. The example given in [14] is for the processing of an invoice in a purchasing department. The TPs for this example are:

- Record the arrival of an invoice.
- Verify that the goods have been received.
- Authorize payment.

Three tasks for this example might be data entry clerks, purchasing officers, and supervisors. Only a data entry clerk could record the arrival, only a purchasing officer could verify receipt, and only a supervisor could authorize payment. A special form of Clark-Wilson policy, called dynamic n-person policies, allow the dynamic assignment of tasks to subjects [14]. In the purchasing department example, any of the available subjects could record the arrival of an invoice, thus becoming the data entry clerk for a particular situation.

In [20] we showed that Clark-Wilson, as adapted to UNIX, can be implemented on DTOS and would expect other task-based access control policies to pose no special problems. Each scenario defines a security policy. At the start of a situation, the system adapts to the appropriate policy. Each task corresponds to a security context. A subject is linked to a task by mapping its security identifier to the context for that task. The policy defines the set of actions allowed for each task and object pair.

Section 6

Fault Tolerance and Security

We originally planned to use a framework being developed by SRI to construct an example policy mapping and its inverse and to show how auditing would be used to recover when reverting from a temporarily-enforced, less-restrictive policy. However, the result of their work [7] was a framework for resolving conflicts among critical system properties, rather than for mapping between policies before and after a policy switch occurs. Among the conflicts studied was types of faults tolerated on a fault-tolerant system and the overhead of fault tolerance [6]. Also, the issue of auditing for recovery is already covered as part of Section 3, DTOS Updates.

An interesting question related to the work in [7], which we explore in this section, is how fault tolerance and security interact. A service can be made fault tolerant by replicating it on several nodes that fail independently. Types of faults that can be tolerated include [3] *crash* in which a node stops communicating with its environment when it fails, *omission* in which messages between nodes are lost, and *Byzantine* in which a node or link may behave in arbitrary ways, including sending incorrect results. The model of fault tolerance that we will use is Primary-Backup, which is suitable for crash and omission faults, but not for Byzantine faults. One of the replicas is designated as the primary and the others as the backups. Requests are handled by the primary, with the results sent to the client and each of the backups as shown in Figure 6-3. If the primary stops (it does not respond timely to some request), one of the

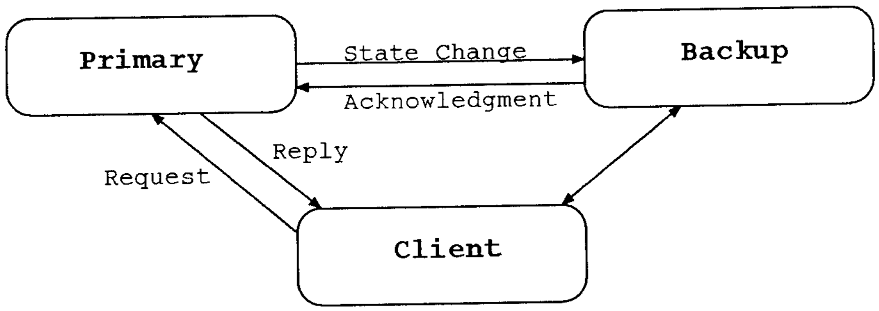

Figure 6-3: Primary-Backup Fault-Tolerance Architecture

backups becomes the primary. The backups acknowledge receipt of messages from the primary to help determine when they crash, allowing a new backup to be started.

The security policy for a system node determines which requests are accepted and which are rejected. It can be defined as the sequences of requests that will be accepted. For a fault-tolerant service, requests are sent to the primary, which decides whether to accept the request. The primary would not send a notice to the backups for those requests that are rejected. Thus, the system security policy is the same as that of the primary.

As long as all nodes on the system have the same security policy, switching to a new primary has no effect on the system security policy. Whether or not a request is accepted is independent of the node that provides the primary. Note that part of this requirement is that clients of the service have equal access to all service nodes; if the communications service does not allow a client to send a request to the primary node, that request is effectively rejected.

6.1 Security Policy Differences as Faults

The relationship between fault tolerance and adaptive security policies results from nodes that have different security policies. A distributed system might span nodes belonging to different organizations, each with its own policy. At the time that a request for service is made, the policy of the primary may differ from that of the client or a backup. When a backup becomes the primary, the policy of that service might change and thus appear to adapt. Alternatively, when the security policy for a distributed system adapts as the result of a change in the environment, there will be a transition period during which nodes might have different policies.

Requests arriving at the primary are either accepted or rejected. For those requests that are accepted, any state changes must be sent to each backup so that its state remains consistent with the primary in case the primary fails and the backup becomes the primary. The security policy for the backup must allow state change information from the primary, even if the backup would have rejected the request. Thus, there could be a flow of information from the client to the primary to the backup that violates a local security policy prohibiting a flow of information from the client to the backup.

If the backup refuses a state update, either the primary or the backup must be considered to have the wrong state and to have failed. One way to choose between these alternatives is to test whether some minimal number **c** of backups concur with the primary by acknowledging the state change within some time interval; if not, the primary has failed and otherwise the nonconcurring backups have failed. Thus, if the primary has sufficient support for its policy, backups with a different policy are removed and the system policy will become consistent, even following a failure of the primary. The algorithm for the primary is shown in Figure 6-4.

```
do forever
   receive request
   if request violates security policy then reject
   else process request
        if state changed then
             broadcast request and change to backups
             wait an appropriate time interval
             if fewer than c acknowledgments then stop fi
             mark backups that did not acknowledge as failed
        fi
        send response to client
   fi
od
```

Figure 6-4: Primary Server

The value of **c** should be less than half of the total number of active nodes. Otherwise, thrashing could occur between two policies. For example, consider a service with a primary and two backups and for which there are two security policies. If **c** is 2 (and therefore is greater than 3/2), the primary fails if either backup does not concur with the acceptance of a request. After choosing a new primary and starting a new backup, complete agreement on the policy may still not occur, leading to a sequence of failures. If **c** is 1, the primary needs concurrence by only one of the backups (a majority vote). If it does not have such an agreement and fails, the backup that is chosen as the new primary will agree in policy with the remaining backup and will be in the majority. A corollary of this principal is that if there is a single backup, **c** is 0 and the backup fails if it disagrees with the primary.

The algorithm in Figure 6-4 uses a blocking protocol in which the primary does not send a response to the client when it fails, since nodes in the Primary-Backup model fail by crashing. If there is only a single backup, this algorithm can be optimized to use a nonblocking protocol by sending the response before sending the state changes and waiting for the acknowledgment.

The primary in Figure 6-4 does not inform the backups when it rejects a request or accepts a request that does not change the state of the service. The primary therefore defines the policy for these cases. An alternative approach is for the primary to notify the backups in these cases, thus permitting the backups to withhold concurrence and cause the primary to fail.

6.2 Policy Transitions

One occasion when the policies of the primary and the backups can be expected to differ is while the system policy is adapting. A policy change broadcast across a distributed system will arrive at some nodes before others. Also, some nodes will process the change more promptly than others. One situation is where the primary begins using the policy change after the arrival of a request, but the backups begin using it before they receive the state changes due to the request. The reverse situation, where the primary begins using the policy change before the arrival of the request, but the backups begin using it after the state changes, is also possible.

For the first situation, shown in Figure 6-3, assume that old policy allows the request but the new policy does not. Since the request arrives before the policy

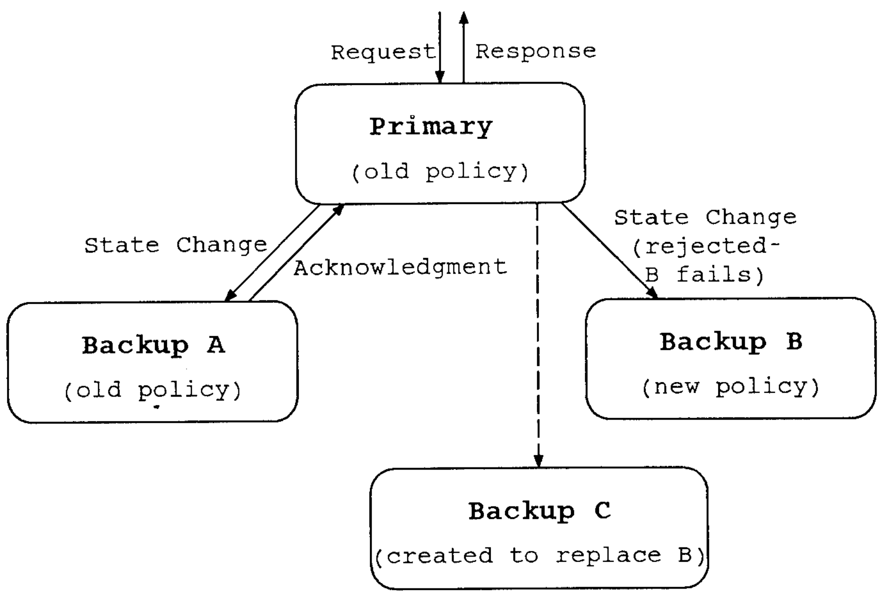

Figure 6-5: Situation 1: Primary uses Old Policy

change, the primary will accept and process the request. When the state change is sent to the backups, if at least **c** backups have not yet received the new policy and therefore acknowledge the state change, those backups that have received the new policy will fail. In Figure 6-3, assuming that **c** is 1, Backup B's state becomes inconsistent with the service state as maintained by the Primary and

Backup A and therefore B fails. What this means is that the service policy is still the old policy and these failed backups used the new policy too soon. Eventually, the new policy will reach the primary and the other backups and the service policy will change. If, however, the backups have received the new policy and do not acknowledge the state change, the primary will fail. In this case, the service policy has changed to the new policy and the primary tried to use the old policy too long. In Figure 6-3, if Backup A was using the new policy, it also would reject the state change, Primary would fail instead of Backup B, and either Backup A or Backup B would become the primary.

For the second situation, assume that new policy allows the request but the old policy does not. This situation covers the case where the 'old policy' and the 'new policy' labels in Figure 6-3 are switched. If sufficient backups accept the state change, the service is enforcing the new policy and backups that fail tried to use the old policy too long. Otherwise, the primary tried to use the new policy before it became the service policy.

This procedure generally serializes the processing of requests with policy changes for a service. The service will continue to use the old policy until the primary and a sufficient number of backups use the new policy. Nodes that are out of step with the service policy will be considered to have failed. However, the second situation above can lead to a serialization failure if the primary using the new policy fails and is replaced by a backup using the old policy. For example, consider a service with a primary and two backups with **c**=1 such that the primary and one of the backups is using the new policy and the other backup is using the old policy. A request is accepted by the primary and the state change is acknowledged by the backup using the same policy as the primary but rejected by the other backup. The rejecting backup has therefore failed and is replaced by another backup which also uses the old policy. Next, the primary fails, the backup using the old policy becomes the primary, and a new backup that uses the old policy is started. The policy has reverted back to the old policy.

Section 7 Conclusion

This work demonstrates that a client-server architecture, such as DTOS, is suitable for use in an adaptable secure environment. The facilities needed for adaptive policies provided by this architecture include:

- A very general enforcement mechanism is separate from the definition of the policy. Thus, a change in the policy requires a small change in the system that can be accomplished by either loading a new table or switching to a new Security Server.
- The ability to cause the policy to change is restricted by making the changes subject to the security policy.
- Any cached security information can be flushed.
- The security attributes of subjects and objects are represented only by an identifier outside of the Security Server. The attributes for a particular entity can be changed, or even replaced by a new set of attributes, by changing the mapping between security identifiers and security contexts in the Security Server.
- Mechanisms for auditing can be separated from auditing policy in a manner similar to the separation used for security. Thus, the events that are audited can be made to depend on the security policy.

Many of these facilities were already present in DTOS. We added facilities for switching between Security Servers, signaling servers other than the microkernel that a policy switch has occurred and that any cached security information should be flushed, auditing, and flushing page table protections. Most of these changes are being added to the released version of the DTOS system and therefore may be tested in additional ways.

7.1 Open Issues

We have identified several ways in which this work could be extended:

Additional Policies. The current contract has demonstrated that a server-client architecture that separates policy from enforcement is suitable for adaptive policies and it has influenced several changes in the DTOS design. However, the policy changes tested are small and constrained to MLS with Type Enforcement. As the prototype is used by other organizations, additional policies are being implemented and could be used to provide more extensive tests. Possibilities include a task-based access control policy and an MLS policy in which levels are collapsed.

Recovery from a Relaxation Policy. A related project is to extend a Security Server to recover from a relaxation policy, making use of the audit log that we produce. The studies by ORA and SRI talk about such recovery, but the details need to be fleshed out and tried. We put auditing in place during our current work, but did not provide any analysis of the audit log and the use of the results in a Security Server.

Delegation Policies. Another extension to the Security Server would be delegation in which the policy is adapted to temporarily tighten security for particular subjects. This requires a Security Server request to register a monitor for these subjects, and a change to the Security Server so that it consults the monitor. To test this facility, a monitor would have to be built.

High-Level Policies. Users of a system think about the security policy in high-level terms, such as 'all mail sent from the system must pass through a filter' or 'only the staff and subordinate commanders are allowed to know the details of the attack plan'. The policy used by the system, however, consists of types of accesses allowed to particular subjects for particular objects. Making the translation can be complicated. For adaptive policies, the situation is worse since several translations are required and also there might be some mandatory constraints that apply to all of the potential policies. Furthermore, satisfying the mandatory constraints may depend on what flows have occurred during previous policies. Tools are necessary to set the policies correctly.

Policy Composition Tool. A particular tool would be one that composed a policy from policies of the various components. Romulus [11] provided a theory about combining components satisfying the same restrictive policy to form a restrictive system. With adaptive policies, however, we envision a system made up of components that satisfy different policies, some of which may change. A change in policy potentially can be modeled as sequential composition of components with different policies. The mandatory constraints form the system policy that the composition satisfies.

This theory needs to be considered further and requirements for such a tool developed.

Effect of Policy Changes on Applications. Undesired results could occur if the policy adapts in the middle of the execution on some application. For example, integrity might be compromised if the policy changes while a trusted application is in the middle of an update. Provisions for a 'critical section', in which certain accesses are guaranteed, could be provided. The implementation of such a critical section along with tests should be performed.

Bibliography

[1] Lee Badger. Providing a flexible security override for trusted systems. In *Proceedings of Computer Security Foundations Workshop III*, pages 115–121, Franconia, NH, June 1990. IEEE.

[2] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings 8th National Computer Security Conference*, pages 18–27, Gaithersburg, MD, October 1985.

[3] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Optimal primary-backup protocols. In *Proceedings of the Sixth International Workshop on Distributed Algorithms*, pages 362–378, Haifa, Israel, November 1992.

[4] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, CA, April 1987.

[5] Todd Fine and Spencer E. Minear. Assuring Distributed Trusted Mach. In *Proceedings IEEE Computer Society Symposium on Research in Security and Privacy*, pages 206–218, Oakland, CA, May 1993.

[6] Li Gong and Jack Goldberg. Implementing adaptive fault-tolerant services for hybrid faults. Technical Report SRI-CSL-94-04, Computer Science Laboratory, SRI International, Menlo Park, CA, September 1994.

[7] Li Gong and Teresa Lunt. Handling critical system requirements in adaptive systems. Technical Report RL-TR-95-100, Rome Laboratory, June 1995.

[8] J. Thomas Haigh et al. Assured service concepts and models: Security in distributed systems. Technical Report RL-TR-92-9, Vol II, Rome Laboratory, January 1992.

[9] Geoffrey R. Hird, Daryl McCullough, Stephen Brackin, and Doug Long. Research advances in handling adaptive security. Technical Report RL-TR-95-92, Rome Laboratory, June 1995.

[10] Paul A. Karger. New methods for immediate revocation. In *Proceedings IEEE Computer Society Symposium on Research in Security and Privacy*, pages 48–55, Oakland, CA, May 1989.

[11] Tanya Korelsky et al. ULYSSES: A computer-security modeling environment. In *11th National Computer Security Conference*, pages 20–28, October 1988.

[12] Keith Loepere. *OSF Mach Kernel Principles*. Open Software Foundation and Carnegie Mellon University, final draft edition, May 1993.

[13] Spencer E. Minear. Providing Policy Control Over Object Operations in a Mach Based System. In *Proceedings of the Fifth USENIX UNIX Security Symposium*, pages 141–156, Salt Lake City, Utah, June 1995.

[14] Michael J. Nash and Keith R. Poland. Some conundrums concerning separation of duty. In *IEEE Symposium on Security and Privacy*, pages 201–207, Oakland, CA, May 1990.

[15] NCSC. Trusted Computer Systems Evaluation Criteria. Standard, DOD 5200.28-STD, US National Computer Security Center, Fort George G. Meade, Maryland 20755-6000, December 1985.

[16] Richard C. O'Brien and Clyde Rogers. Developing applications on LOCK. In *Proceedings 14th National Computer Security Conference*, pages 147–156, Washington, DC, October 1991.

[17] Richard F. Rashid. Mach: A case study in technology transfer. In Richard F. Rashid, editor, *CMU Computer Science: A 25th Anniversary Commemorative*, chapter 17, pages 411–421. ACM Press, 1991.

[18] Edward A. Schneider, D. G. Weber, and Tanja de Groot. Temporal properties of distributed systems. Technical Report RADC-TR-89-376, Vol I, Rome Air Development Center, September 1989.

[19] Secure Computing Corporation. DTOS Formal Security Policy Model (FSPM). Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, June 1995. DTOS CDRL A004.

[20] Secure Computing Corporation. DTOS Generalized Security Policy Specification. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, January 1995. DTOS CDRL A019.

[21] Secure Computing Corporation. DTOS Kernel Interfaces Document. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, April 1995. DTOS CDRL A003.

☆U.S. GOVERNMENT PRINTING OFFICE: 1996-710-126-20233